Introduction to Threads

Steve Karmesin

10 September 1998

Overview

- What are the core ideas?
  - Create, join, mutex, condition variables
- Race conditions
  - The “dangling pointers” of threads.
- Benefits from using threads.
  - Adaptivity, flexibility
- How will POOMA II use threads?
  - Conceptually
- How might Tecolote use threads?
  - High level loops, clusters of threads.

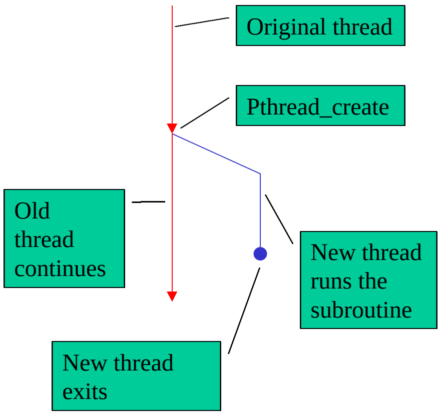
Thread Libraries

- Pthreads
  - Procedural
  - Standard
  - Provided by vendor
  - General
  - Often slow.

- Tulip
  - Object Oriented
  - Java interface
  - Provided by ACL
  - General
  - Fast
  - Can sit on top of pthreads.

Thread Creation

- pthread_create
  - thread id pointer
  - attributes
  - function pointer
  - data pointer
- Creates a new thread.
- Thread runs the function, and the caller continues.
- Thread exits when the function returns.



Original thread

Pthread_create

Old thread continues

New thread runs the subroutine

New thread exits

Write To Disk

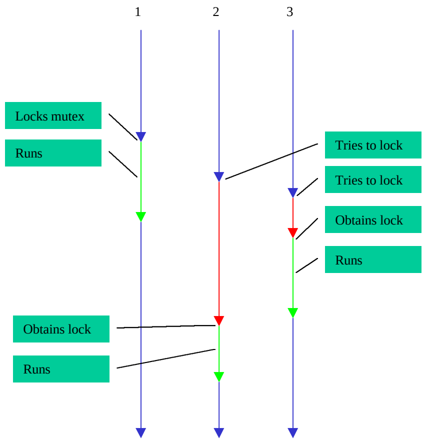
- Generate a thread to asynchronously write data to disk.
- Open the file
- Write the data
- Close the file
- No synchronization is needed.
- Allows the main thread to go on while the disk I/O is happening.
- Also need to delete the data structures.

```
struct data_t {
  char *file;  double *p;  int n;
};

void *dump(void *vp)
{
  data_t *p = vp;
  FILE *fp = fopen(p->file,”w”);
  fwrite(p->p,sizeof(double),p->n,fp);
  fclose(fp);
}
main()
{
…
   pthread_t id;
   pthread_create(&id,NULL,dump,datap);
…
}
```

Mutex

- Only one thread can lock a mutex at one time.
- Anyone else trying to lock it will wait.
- `pthread_mutex_t mutex;`
- `pthread_mutex_lock(&mutex);`
- `pthread_mutex_unlock(&mutex);`
- Waiting on a mutex often means spinning
- Should usually only try to lock mutexes when you expect to get it, but want to be safe.



1

2

3

Locks mutex

Runs

Tries to lock

Tries to lock

Obtains lock

Runs

Obtains lock

Runs

Bad Barrier

```
// Barrier for n threads.
Barrier(int n)
{
   static int count = 0;
   if (count==0)
      count = n-1;
   else
      count -= 1;
   while (count > 0) ;
}
```

- What is supposed to happen
  - First thread sets count =n-1
  - Next threads decrement count
  - Everybody waits until all decrements are done.

- Lots of ways this can go wrong if multiple threads enter at about the same time.
  - Multiple threads see count==0 before one writes count=n-1
  - count -= 1 is really read, subtract, store, and multiple threads could read before any write.
- These are classic race conditions.

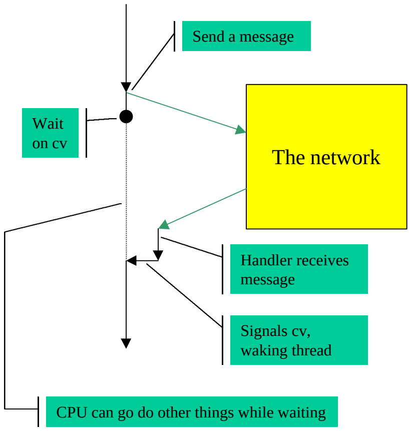
Better Barrier Using Mutex

```
// Barrier for n threads.
Barrier(int n)
{
   static int count = 0;
   static pthread_mutex_t
      mutex=PTHREAD_MUTEX_INIT;
   pthread_mutex_lock(&mutex);
   if (count==0)
      count = n-1;
   else
      count -= 1;
   pthread_mutex_unlock(&mutex);
   while (count > 0) ;
}
```

- Fixes the problems noted in the previous.
  - Only one thread can be in the section that modifies count at one time.
- There is still a problem though…
  - Two barriers in a row.
  - Count goes to zero, some threads go into the next barrier and sets count again before all the threads leave the while loop.

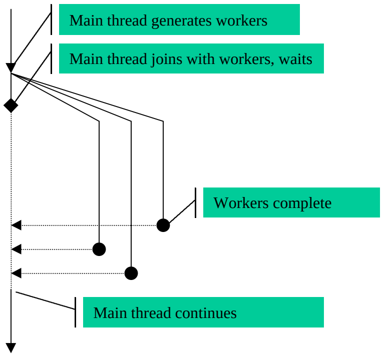
Condition Variable

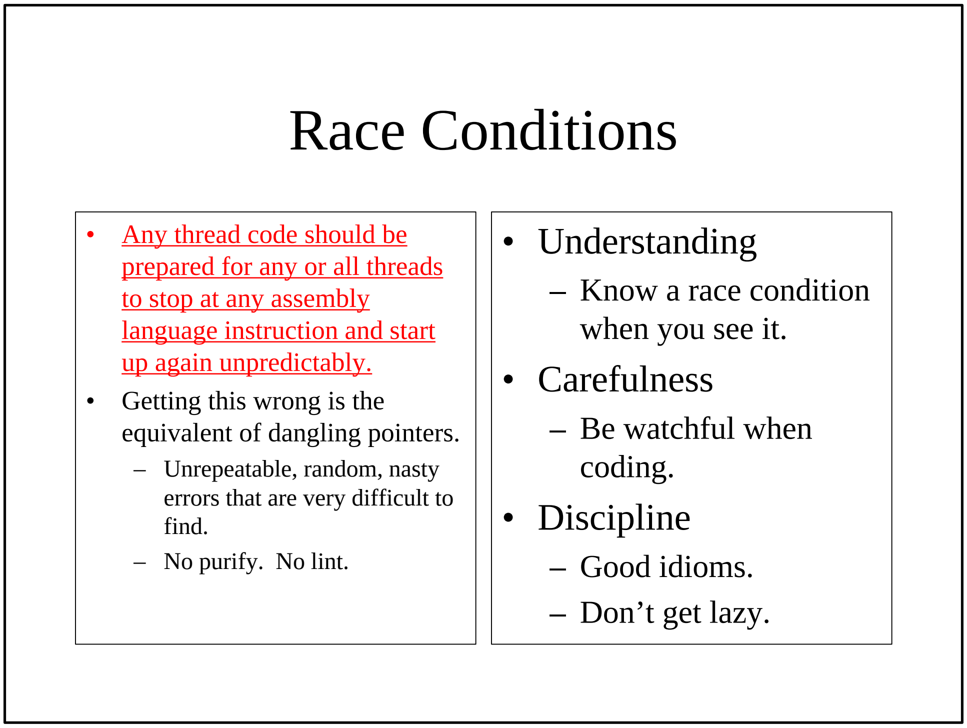
- “Wake me up when something happens”
- Useful when you think something will happen someday and you want to wait until then.
- One or more threads wait.
- One signals them.
- Exact syntax is fussy, have to use a mutex to lock access.


Send a message
Wait on cv
The network
Handler receives message
Signals cv, waking thread
CPU can go do other things while waiting

Join

- Wait for another thread to finish.
- Useful when you generate a bunch of workers, and want to wait until they all fiinish.
- Join waits for one thread at a time.
- To wait for N threads, join one at a time.

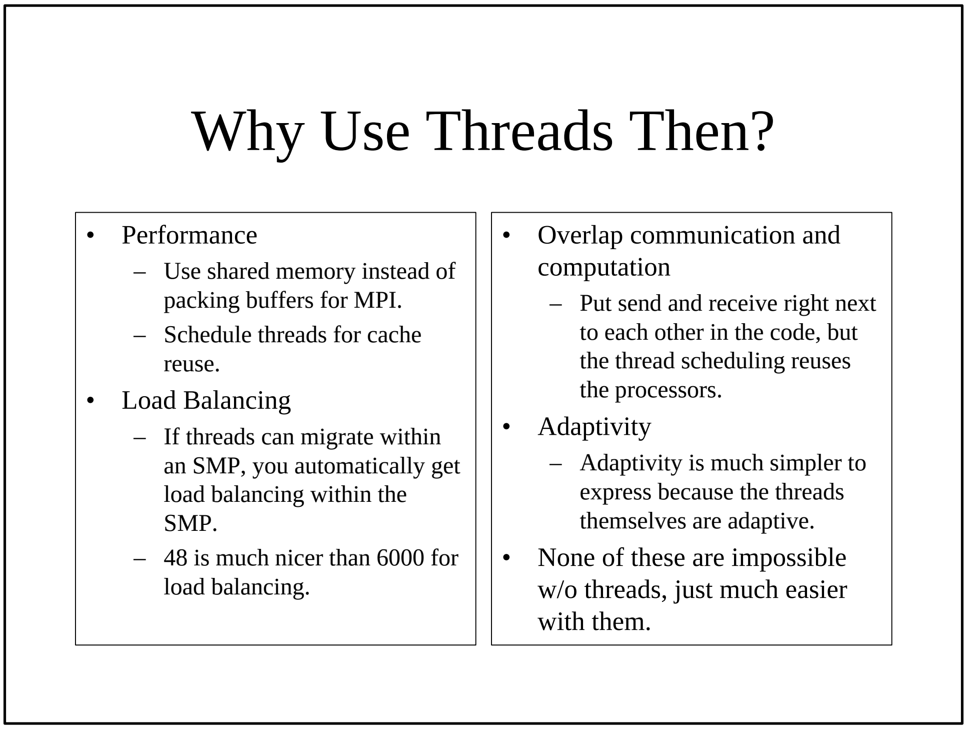

Main thread generates workers
Main thread joins with workers, waits
Workers complete
Main thread continues



Race Conditions

- Any thread code should be prepared for any or all threads to stop at any assembly language instruction and start up again unpredictably.
- Getting this wrong is the equivalent of dangling pointers.
  - Unrepeatable, random, nasty errors that are very difficult to find.
  - No purify. No lint.

- Understanding
  - Know a race condition when you see it.
- Carefulness
  - Be watchful when coding.
- Discipline
  - Good idioms.
  - Don’t get lazy.



Why Use Threads Then?

- Performance
  - Use shared memory instead of packing buffers for MPI.
  - Schedule threads for cache reuse.
- Load Balancing
  - If threads can migrate within an SMP, you automatically get load balancing within the SMP.
  - 48 is much nicer than 6000 for load balancing.

- Overlap communication and computation
  - Put send and receive right next to each other in the code, but the thread scheduling reuses the processors.
- Adaptivity
  - Adaptivity is much simpler to express because the threads themselves are adaptive.
- None of these are impossible w/o threads, just much easier with them.

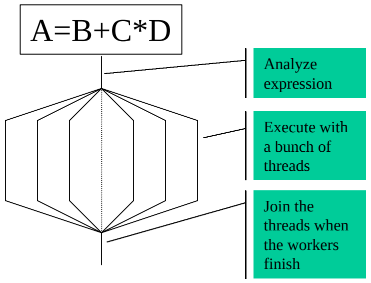
POOMA II Internal Threads

- POOMA sees the world as a series of expressions to evaluate.
  - A=B+C*D
- POOMA I breaks the expression down into pieces (vnodes) and evaluates them on every CPU.
- One “parse thread” per CPU (per MPI process).

- POOMA II also sees the world in terms of expressions to evaluate.
- POOMA II also breaks the expressions down into pieces for evaluation.
- It then hands those pieces off to threads for evaluation.
- One parse thread per a cluster of CPU’s, perhaps one per SMP.
- Many worker threads.

Lock-Step Implementation

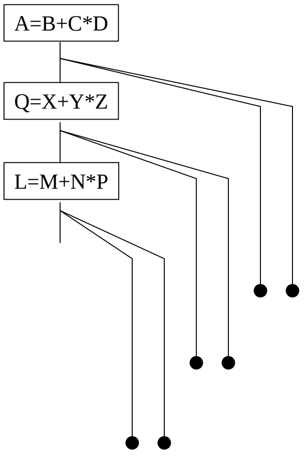
- Suppose we did thread create and join for each expression.
- A=B+C*D would loop over vnodes, could generate a thread per vnode, then wait until they all finish (using join).
- Advantages over current:
  - Load balancing on box
  - Possible overlap of GC fill.
- Disadvantages over current:
  - Thread creation and destruction time.


A=B+C*D
Analyze expression
Execute with a bunch of threads
Join the threads when the workers finish

Each line finishes before the next one starts.

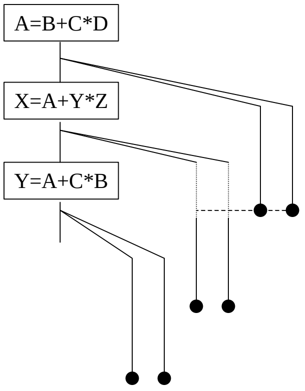
Overlapping Implementation

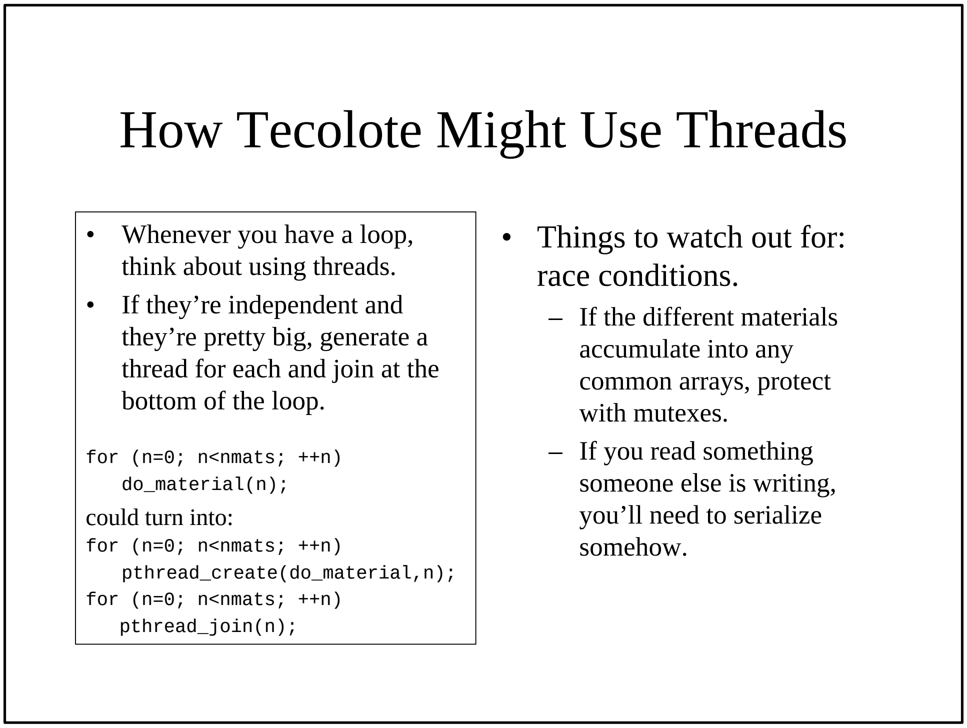
- Instead of joining at the end of each statement, allow more overlap.
- “Parse thread” runs through the user code.
- Generates worker threads.
- Parse thread could get way ahead of workers.
- If some workers need communication, they wait, and something else runs.
- Auto-load balance inside SMP.



Data Dependencies

- Previous example had no data dependencies.
- With data dependencies, some threads can’t run until others finish.
- Can handle this with condition variables (POOMA II actually does something fancier…)
- Threads sleep until their data is available.
- Schedule threads to use data that has just been released.





How Tecolote Might Use Threads

- Whenever you have a loop, think about using threads.
- If they're independent and they're pretty big, generate a thread for each and join at the bottom of the loop.

```
for (n=0; n<nmats; ++n)
   do_material(n);
```

could turn into:

```
for (n=0; n<nmats; ++n)
   pthread_create(do_material,n);
for (n=0; n<nmats; ++n)
   pthread_join(n);
```

- Things to watch out for: race conditions.
  - If the different materials accumulate into any common arrays, protect with mutexes.
  - If you read something someone else is writing, you'll need to serialize somehow.